NBSIR 78-1551

Access Functions for Packed Scatter Tables

Bruce E. Martin

Systems and Software Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D.C. 20234

August 1978

U.S. DEPARTMENT OF COMMERCE

NATIONAL BUREAU OF STANDARDS

NBSIR 78-1551

ACCESS FUNCTIONS FOR PACKED SCATTER TABLES

Bruce E. Martin

Systems and Software Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D.C. 20234

August 1978

NBSIR 78-1551

ACCESS FUNCTIONS FOR PACKED SCATTER TABLES

Bruce E. Martin

Systems and Software Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D.C. 20234

August 1978

TABLE OF CONTENTS

DISCLAIMER

The table functions described in this report have been written and tested carefully. The possibility of improper application requires that NBS expressly disclaim any and all consequences of using the functions. A particular warning must be issued on attempts to speed table access by storing indices: such schemes are guaranteed failure, since items move with new insertions.

Access Functions for Packed Scatter Tables

Bruce E. Martin

Three PASCAL access routines are given for packed scatter tables. INSERT packs tables of integer keys; FIND retrieves the keys; DELETE deletes keys. While the routines currently access integer keys (for use in performance testing with pseudo - random integers), they can easily be converted to access other data types -- character strings in a symbol table, for example. To enhance portability, the code is straightforward PASCAL without any input-output capabilities. Appendices contain specific comments on the routines, listings, and sample results.

SUMMARY OF THE ACCESS METHOD

Hashing techniques allow insertion and retrieval of keys by computing a function h(key) and storing the key in T[h(key)], where T is an indexed table. Since the function h(key) generally does not compute unique addresses, different keys may "collide". Simple collision-resolving methods search for an alternate location to store the key being inserted. But such methods can cause slow retrieval of keys. More sophisticated methods search for alternate locations for other keys as well as for the key being inserted. INSERT uses a generalized collision-resolving method that recursively considers table rearrangement. The user controls the method to be used with the parameter DEPTH. Lyon gives a more detailed discussion of the algorithm in [1]. A summary of the method appears here. Specific comments on the PASCAL code appear in appendix A.

Integer keys are inserted in the scatter table by first calculating a table address using a primary hash function (key mod tablesize). If the slot in the table addressed by the primary hash function is not occupied, the key is inserted. Otherwise, the cost of displacing the contents of the slot is calculated by calling the function DISPLACE. Next, the cost of displacing the key is calculated by a second call of DISPLACE. DISPLACE returns a stack that indicates how the table should be rearranged. The table is rearranged using the stack returned by the call of DISPLACE with the least displacement cost.

The function DISPLACE makes probes into the table by increments of the secondary hash step (key mod (tablesize-2) +1) until an open slot is found. Each slot probed during the search for an open slot is considered for displacement by recursive calls of DISPLACE. The recursion terminates when the deepest level of recursion, as specified by the user with the parameter DEPTH, is reached or whenever displacement of further slots cannot possibly find a better solution. The deepest call of DISPLACE returns the additional penalty to probe to the free slot; that is, PENALTY(number of probes to free slot) minus PENALTY(number of probes to table address), where PENALTY is a forcing function defined by the user (see [1]). The rearrangement stack returned by the deepest call of DISPLACE consists of the table address of the item being displaced and the address of the free slot. At higher levels, the total displacement cost of each subsequent slot is calculated to be the additional cost to probe to the slot plus its displacement cost, and the minimum is returned. The stack at higher levels of DISPLACE consists of the address of the item being displaced and the stack returned from the chosen call of DISPLACE.

A table of counters of search lengths is maintained for faster rejection of keys not in the table. As keys are inserted, the number of probes to find each key is recorded by incrementing a counter in the search length counter table, KICKOUT, where the search length is the index of KICKOUT. Since key insertion possibly causes other keys to be displaced, thus changing each displaced key's search length, old and new search lengths of displaced keys are also returned on the rearrangement stack to keep KICKOUT updated and maintained. When a key is deleted, the appropriate search length counter is decremented.

Keys are retrieved by first calculating their original primary hash function and secondary hash step. Next, the table is probed until the key or an open slot is found or the number of slots probed equals the maximum search length. Generally, the performance of retrieving keys in the table does not improve as keys are deleted. For example, a table half filled performs much better than a table that is first completely filled and then has half of its keys deleted. Rejecting keys not in the table improves as keys are deleted provided deletions cause the maximum search length to decrease. Again, the half-filled table performs better than the table that is half-deleted. However, as keys are reinserted, both rejection and acceptance improve because insertion of keys causes the table to be rearranged more optimally. See Appendix C for measurements that are typical of correctly executing functions: Testing of the routines on a new system should give similar results.

Declarations

The following must be declared by the calling program:

```
const
  tablesize    {the size of the table into which keys
                will be inserted.  Must be prime!}
  maxreal       {largest real for particular instal-
                 lation.}
  kicksize      {size of the search counter table}*

type
  table=array[0..tablesize-1] of integer;
  kicktab= array[-1..kicksize+1] of integer;
  stkptr = ^stkelmnt;
  stkelmnt = record
               ind, oldlen, newlen: integer;
               next: stkptr
             end;

var
  oldnodes: stkptr;     {for node (de)allocation}
```

* Kicksize should be the expected longest probe for a particular depth, penalty function and table filling. If the estimate of the maximum search length is too small, the rejection performance of the table may deteriorate. Therefore, a generous estimate of kicksize is desirable.

Initializations

The following must be initialized by the calling program:

VARIABLES	WHEN	INITIALIZED TO
of type TABLE	for new table	-1
OLDNODES	first use of routines	nil

of type KICKTAB   for new table   as follows:

```
          kickout[-1]:=0; {no overflow with new table}
          kickout[0]:=1;  {maximum probes with new table}
          kickout[1..kicksize+1]:=0;
```

Node (De)allocation

Since node (de)allocation differs from one PASCAL installation to the next, the INSERT routine, in the interest of portability, explicitly controls node (de)allocation for its rearrangement stack via procedures GETNODE and FREENODE. The procedures use a global variable OLDNODES, which points to a linked list of nodes that grows and shrinks during execution.

REFERENCES

[1] Lyon, G.E. "Packed Scatter Tables", Comm. of the ACM (tentative October, 1978)

[2] Lyon, G.E. "Batch scheduling from short lists", Information Processing Letters, (to appear)

APPENDIX A: Documentation of PASCAL code

Procedure insert

PARAMETER	TYPE	
tab	table	the table in which keys are inserted.
key	integer	the integer to be inserted.
depth	integer	depth of recursion for displacement.
kickout	kicktab	table of counters of search lengths. KICKOUT[1..KICKSIZE] are counters of search lengths, where the length is the index in KICKOUT. KICKOUT[Ø] = longest search length. KICKOUT[-1] = longest search length if an overflow occurs. KICKOUT[KICKSIZE+1] is counter of overflow search lengths. KICKOUT[Ø] traps to KICKSIZE+1 if overflow ocurs.

VARIABLE	TYPE	
index	integer	primary hash index. TAB[INDEX] is considered for displacement if a collision occurs.
temp	integer	stores the contents of TAB[INDEX] while displacement of the new key is being considered.
lenl	integer	length of the longest search returned when TAB[INDEX] is considered for displacement.

len2	integer	length of the longest search returned when the key is considered for displacement.
cost1	real	cost of displacing TAB[INDEX].
cost2	real	cost of displacing the key.
stk1	stkptr	rearrangement stack returned when TAB[INDEX] is considered for displacement.
stk2	stkptr	rearrangement stack returned when the key is considered for displacement.

LINE NUMBER(S)

142..144

The primary hash index is calculated. Another key has search length of 1 so KICKOUT[1] is incremented. If the slot is empty or marked deleted, the key is inserted and INSERT is exited. -1 indicates empty slots; -2 marks deleted slots;

148..149

The cost of displacing TAB[INDEX] is calculated provided DEPTH > Ø. DEPTH=Ø means the key should be inserted in the first free slot and no displacements occur.

15Ø..152

The cost of displacing the key is calculated by temporarily storing TAB[INDEX] in TEMP. The key is inserted in tab[index] and DISPLACE is called. This was designed so that DISPLACE would have the table address as an parameter (necessary for recursive calls) and so INSERT would have the key as a parameter, making table addresses invisible to the user. Note that with the second call of DISPLACE, COST1 is the actual parameter corresponding to the DISPLACE formal parameter MAX. This keeps the second call of DISPLACE from considering any displacements that are more costly than the displacement found by the first call of DISPLACE.

153..162

If both the key and TAB[INDEX] were considered for displacement, the table is rearranged by REARRANGE according to the stack returned by the call of DISPLACE returning the lower cost. If only the key was considered for displacement, the table is rearranged by STK2. Finally, after table rearrangement, both stacks are deallocated by FREENODE.

procedure getnode

7..15

A node is allocated from OLDNODES or by the pascal function NEW.

procedure freenode

16..27

A linked list is walked and deallocated to OLDNODES.

procedure rearrange

30

If the search length passed to it is greater than the current maximum in KICKOUT[0], KICKOUT[0] is updated.

31..47

Each node in the stack has four fields: IND,OLDLEN,NEWLEN and NEXT. For each node: a) OLDLEN and NEWLEN are tested for overflow. If so, KICKOUT[0] traps to the overflow counter. b) KICKOUT[NEWLEN] is incremented and KICKOUT[OLDLEN] is decremented. c) the contents of TAB[NEXT^.IND] are moved to TAB[IND]. d) the next node of the stack is used. When the last node is encountered, the key is moved to TAB[IND].

function displace

PARAMETER	TYPE	
index	integer	address of item to be considered for displacement.
depth	integer	depth of recursion for which displacement should be considered.
max	real	MAX is an upper limit on cost for displacement consideration. It is the best solution found so far at higher levels of recursion.
rjstack	stkptr	contains indicies of slots rejected at higher levels of recursion. A slot which is rejected at a higher level of recursion will not lead to a better solution at a deeper level.
stack	stackptr	returns the rearrangement stack for best solution at a given level. (var parameter)
length	integer	returns the length of the longest search. (var parameter)

VARIABLE	TYPE	
ind	integer	used to calculate subsequent slots in the table.
probetoind	integer	number of probes to hash to TAB[INDEX].
probetofree	integer	number of probes to the first free slot.
counter	integer	slot counter. Used in calculating subsequent locations to probe the

		table.
step	integer	equal to the secondary hash function for probing.
hashl	integer	equal to the primary hash function for probing.
next	integer	address of the next slot.
srchlen	integer	longest search from deeper levels of recursion.
uplim	real	the upper limit on displacement cost at a given level. UPLIM is the minimum of the cost to move TAB[INDEX] to a free slot and MAX (the least cost found at higher levels).
totcost	real	additional cost of probing to next slot plus cost of displacing next slot.
pentonext	real	additional cost of probing to next slot, that is PENALTY(probes to next slot) - PENALTY(probetoind).
thisnode	stkptr	pointer to node pushed on stack. It is used to update the new search length.
savrj	stkptr	saves a copy of rjstack upon first execution of DISPLACE.
bestack	stkptr	saves the stack returned by DISPLACE returning the lowest cost. BESTACK is in turn returned to higher levels of DISPLACE.
tstack	stkptr	temporary stack for calls of

DISPLACE.

LINE NUMBER(S)	
84..95	The primary and secondary hash functions are calculated. Slots are probed to find the number of probes to INDEX and a free slot.
96..100	INDEX, number of probes to index, number of probes to free slot are pushed on STACK as IND, OLDLEN and NEWLEN, respectively. NEWLEN may have to be updated later if a better solution is found. Therefore, THISNODE saves the node. So far, the best solution found is to move TAB[INDEX] to a free slot so BESTACK is set to STACK. RJSTACK is saved.
101..103	The tentative longest search is PROBETOFREE so LENGTH defaults to PROBETOFREE. The index of the free slot and two dummy constants are pushed on BESTACK. UPLIM is the additional cost of probing to the free slot.
104	If DEPTH=0, no subsequent slots are to be considered for displacement. The recursion has terminated. The stack with INDEX and the address of the free slot is returned. The value of DISPLACE is UPLIM. Otherwise, subsequent slots are considered for displacement.
105..109	If a better solution than UPLIM was found from a higher level of recursion then UPLIM is updated. The primary hash location is the first to be considered for relocation. The cost to probe to the first location is PENALTY(1) - PENALTY(probetoind).
110	While the cost to probe to the next slot is greater than UPLIM, the following is done:

112..135

If the next slot has not already been considered, it is considered for relocation. If the total cost, TOTCOST, is lower than the current lowest cost UPLIM, a better solution has been found and UPLIM, BESTACK and LENGTH are updated to TOTCOST, TSTACK and SRCHLEN, respectively. Otherwise, the slot being considered for relocation is pushed on the reject stack, RJSTACK. The next slot to be considered and the additional cost to probe to it are calculated.

136..140

The value of DISPLACE returned is the UPLIM. BESTACK is returned as the rearrangement stack.

procedure _push_

PUSH gets a new node and pushes the arguments I,OLDLEN,NEWLEN onto the stack.

function _member_

MEMBER returns true if its integer argument is a member of the stack. Otherwise it returns false.

function _penalty_

PENALTY is a forcing function defined by the user. Currently it is linear; it returns its argument. To force insertion of keys in a different manner the user must change the forcing function.

function _find_

FIND tries to find the key in the table. If found, its table location is returned, otherwise -1 is returned. The variable probes returns the number of probes to find (or reject) the key.

procedure _delete_

DELETE finds the key in the table by calling function FIND, deletes the key

and updates KICKOUT accordingly. A slot
is marked deleted by setting it to -2.

APPENDIX B: Listings of access routines

```
procedure insert (var tab: table; key, depth: integer;
                  var kickout:kicktab);

var
  index, temp, len1, len2: integer;
  cost1, cost2: real;
  stk1, stk2: stkptr;

  procedure getnode(var p:stkptr);
   begin
    if oldnodes=nil then new(p)
     else begin
      p:=oldnodes;
      oldnodes:=oldnodes^.next;
     end;
    p^.next:=nil
   end;

  procedure freenode (first,last:stkptr);
  var
    x:stkptr;
   begin
    if first<>last then
     begin
      x:=first;
      while first^.next<>last do first:=first^.next;
      first^.next:=oldnodes;
      oldnodes:=x
     end
   end;

  procedure rearrange (stack: stkptr; key, length: integer);
   begin
    if length > kickout[0] then kickout[0]:=length;
     repeat
      if stack^.newlen > kicksize then
       begin
        if kickout[-1] < stack^.newlen then
          kickout[-1]:=stack^.newlen;
        kickout[0]:=kicksize+1; { trap to overflow counter }
        stack^.newlen:=kicksize+1
       end;
      if stack^.oldlen > kicksize then
         stack^.oldlen:=kicksize + 1;
      kickout[stack^.newlen]:=kickout[stack^.newlen] + 1;
      kickout[stack^.oldlen]:=kickout[stack^.oldlen] - 1;
      if stack^.next <> nil then
```

```
           tab[stack^.ind]:=tab[stack^.next^.ind]
        else tab[stack^.ind]:=key;
        stack:=stack^.next
       until stack=nil;
      while kickout[kickout[0]] = 0 do kickout[0]:=kickout[0]-1;
     end;

    function displace (index,depth:integer; max:real; rjstack:stkptr;
                       var stack: stkptr; var length: integer):real;

    var
      ind, probetoind, probetofree, counter, step, hashl,
      next, srchlen: integer;
      uplim, totcost, pentonext: real;
      thisnode, savrj, bestack, tstack: stkptr;


      procedure push (i,oldlen,newlen:integer; var stack:stkptr);
      var
        node: stkptr;
       begin
        getnode(node);
        node^.ind:=i;
        node^.newlen:=newlen;
        node^.oldlen:=oldlen;
        node^.next:=stack;
        stack:=node
       end;


      function member (i:integer; stk: stkptr):boolean;
      var
        found: boolean;
       begin
        found:=false;
        while (stk <> nil) and (not found) do
          if stk^.ind=i then found:=true
            else stk:=stk^.next;
        member:=found
       end;

      function penalty(i:integer): real;
       {to be defined as desired; currently linear}
       begin
        penalty:=float(i)
       end;

     begin { function displace }
      step:=(tab[index] mod (tablesize-2))+1;
      hashl:=tab[index] mod tablesize;
      probetoind:=0;
       repeat
```

```
        ind:=(hashl + probetoind * step) mod tablesize;
        probetoind:=probetoind + 1
       until ind=index;
      probetofree:=0;
       repeat
        ind:=(hashl + probetofree * step) mod tablesize;
        probetofree:=probetofree + 1
       until (tab[ind]=-1) or (tab[ind]=-2);
      push(index,probetoind,probetofree,stack);
      thisnode:=stack;
      tstack:=stack;
      bestack:=stack;
      savrj:=rjstack;
      length:=probetofree;
      uplim:=penalty(probetofree)-penalty(probetoind);
      push(ind,1,1,bestack);
      if depth > 0  then
       begin
        if uplim > max then uplim:=max;
        counter:=0;
        next:=hashl;
        pentonext:=penalty(1) - penalty(probetoind);
        while uplim > pentonext do
         begin
          if (not member(next,stack)) and (not member(next,rjstack))
          then
           begin
            totcost:=pentonext + displace(next, depth-1, uplim-pentonext,
                                          rjstack, tstack, srchlen);
            if totcost < uplim then
             begin
              uplim:=totcost;
              freenode(bestack,stack);
              bestack:=tstack;
              thisnode^.newlen:=counter+1;
              if 1+counter > srchlen then length:=1+counter
                else length:=srchlen;
             end
            else begin
              push(next,1,1,rjstack);
              freenode(tstack,stack)
             end;
            tstack:=stack;
           end;
          counter:=counter+1;
          next:=(hashl + counter * step) mod tablesize;
          pentonext:=penalty(1+counter) - penalty(probetoind)
         end;
        freenode(rjstack,savrj)
       end;
      stack:=bestack;
      displace:=uplim
```

```
       end;

     begin   { procedure insert }
      index:=key mod tablesize;
      kickout[1]:=kickout[1]+1;
      if (tab[index]=-1) or (tab[index]=-2) then tab[index]:=key
       else begin
        stk1:=nil;
        stk2:=nil;
        if depth>0 then
          cost1:=displace(index, depth-1, maxreal, nil, stk1, len1);
        temp:=tab[index];
        tab[index]:=key;
        cost2:=displace(index, depth, cost1, nil, stk2, len2);
        if (depth =0) or (cost2 < cost1) then rearrange(stk2, temp, len2)
         else begin
          tab[index]:=temp;
          rearrange(stk1, key, len1)
         end;
        freenode(stk1,nil);
        freenode(stk2,nil)
       end
     end;
    { procedure insert }
```

```
function find(var tab: table; key: integer; var kickout: kicktab;
                var probes: integer): integer;

var
  hashl,step,index,limit: integer;

 begin
  probes:=0;
  hashl:= key mod tablesize;
  step:= key mod (tablesize - 2) +1;
  if kickout[0]=kicksize+1 then limit:=kickout[-1]
    else limit:=kickout[0];
   repeat
    index:=(hashl + probes * step) mod tablesize;
    probes:=probes+1
   until (tab[index]=key) or (tab[index]=-1) or (probes=limit);
  if tab[index]=key then find:=index
    else find:=-1
 end;
{ function find }
```

```
procedure delete(var tab: table; key: integer; var kickout: kicktab);

var
  where,probes: integer;

 begin
  where:=find(tab,key,kickout,probes);
  if where = -1 then writeln(output,key,' not found')
   else begin
    tab[where]:=-2;
    if probes > kicksize then probes:= kicksize+1;
    kickout[probes]:=kickout[probes]-1;
    while (kickout[0]<>1) and (kickout[kickout[0]]=0)
      do kickout[0]:=kickout[0]-1
   end
 end;
{ procedure delete }
```

APPENDIX C: Sample results

The table was 98% filled. The following was done for DEPTh = Ø,1,2,3,4,1Ø: 4899 random integers were generated from a linear-congruential formula i:=33Ø9*i+885321 (mod 41943Ø4). The same 4899 keys were retrieved from the table to calculate retrieval performances. 4899 keys not in the table were generated and rejected to calculate rejection performances. This was repeated 18 times.

TABLESIZE:	4999
NUMBER OF KEYS INSERTED:	4899
PERCENT OF TABLE FILLED:	98
DEPTH OF RECURSION:	Ø
PENALTY FUNCTION USED:	LINEAR

TRIAL	LONGEST PROBE	MEAN PROBES	MEAN REJECTION
1	179	4.Ø5429	48.17125
2	154	3.96142	47.8Ø567
3	266	4.ØØ2Ø4	49.21779
4	181	3.93957	48.19289
5	158	4.Ø694Ø	48.14ØØ2
6	4Ø7	3.84Ø57	48.72545
7	21Ø	3.92712	49.15839
8	131	3.84Ø57	45.55154
9	171	3.92733	47.68626
1Ø	17Ø	3.84751	48.78914
11	144	3.96366	47.68279
12	155	3.92998	47.Ø74Ø9
13	22Ø	4.Ø3184	5Ø.97244
14	249	3.96529	48.2Ø759
15	142	3.82384	45.78485
16	264	4.Ø8389	5Ø.35Ø47
17	195	4.Ø3123	48.Ø2571
18	169	3.89957	48.48152
MEAN	198.Ø5	3.95217	48.22322

```
TABLESIZE:                      4999
NUMBER OF KEYS INSERTED:        4899
PERCENT OF TABLE FILLED:          98
DEPTH OF RECURSION:                1
PENALTY FUNCTION USED:        LINEAR

TRIAL    LONGEST PROBE     MEAN PROBES    MEAN REJECTION
-----    -------------     -----------    --------------
  1           20             2.14696         16.64258
  2           15             2.12880         13.07817
  3           26             2.12247         20.30557
  4           20             2.13574         16.49112
  5           21             2.11614         17.29679
  6           18             2.16431         15.17922
  7           20             2.13247         16.63400
  8           25             2.14778         19.72320
  9           20             2.11859         16.67462
 10           20             2.12267         16.68748
 11           20             2.14227         16.52582
 12           21             2.14206         17.05817
 13           19             2.13655         15.84547
 14           19             2.12288         15.99224
 15           27             2.15554         20.89773
 16           19             2.15819         16.01285
 17           21             2.15839         17.33210
 18           18             2.14472         15.43233

MEAN         20.50           2.13870         16.87830
```

```
TABLESIZE:                   4999
NUMBER OF KEYS INSERTED:     4899
PERCENT OF TABLE FILLED:       98
DEPTH OF RECURSION:             2
PENALTY FUNCTION USED:     LINEAR

TRIAL     LONGEST PROBE     MEAN PROBES     MEAN REJECTION
-----     -------------     -----------     --------------
  1             13            1.89855         11.64564
  2             11            1.90936         10.04123
  3             12            1.90304         10.67605
  4             15            1.89895         13.08532
  5             11            1.92284         10.01408
  6             13            1.89548         11.58522
  7             11            1.91181          9.95264
  8             12            1.91202         10.74402
  9             11            1.92304          9.93794
 10             14            1.91712         12.29271
 11             12            1.89712         10.78138
 12             13            1.87711         11.53031
 13             12            1.91855         10.76546
 14             13            1.91835         11.58236
 15             11            1.92876         10.05409
 16             14            1.89875         12.42600
 17             11            1.91345          9.99183
 18             12            1.90814         10.75648

MEAN          12.27           1.90847         10.99237
```

```
TABLESIZE:                     4999
NUMBER OF KEYS INSERTED:       4899
PERCENT OF TABLE FILLED:         98
DEPTH OF RECURSION:               3
PENALTY FUNCTION USED:       LINEAR

TRIAL     LONGEST PROBE     MEAN PROBES     MEAN REJECTION
-----     -------------     -----------     --------------
  1            11             1.83710           9.95407
  2             9             1.84486           8.35211
  3            11             1.83445           9.95733
  4            11             1.81812          10.00489
  5            10             1.81771           9.12349
  6             9             1.80608           8.28373
  7            10             1.84323           9.17962
  8            10             1.81853           9.17350
  9            11             1.82629           9.95611
 10            11             1.84996           9.94284
 11            11             1.85772           9.97958
 12             9             1.83996           8.37170
 13            10             1.81833           9.16084
 14            11             1.83568           9.92794
 15             9             1.81853           8.28965
 16            10             1.83200           9.14064
 17            10             1.81975           9.17228
 18             8             1.81363           7.42416

MEAN          10.05           1.82955           9.18858
```

```
TABLESIZE:                    4999
NUMBER OF KEYS INSERTED:      4899
PERCENT OF TABLE FILLED:        98
DEPTH OF RECURSION:              4
PENALTY FUNCTION USED:      LINEAR

TRIAL    LONGEST PROBE    MEAN PROBES    MEAN REJECTION
-----    -------------    -----------    --------------
  1            8            1.80894         7.42559
  2           10            1.79975         9.11900
  3            8            1.77873         7.45621
  4            9            1.80036         8.31006
  5            9            1.79199         8.26495
  6           10            1.80261         9.11696
  7            8            1.78362         7.44886
  8            9            1.81036         8.28026
  9           10            1.79159         9.16411
 10           10            1.80159         9.17452
 11            9            1.80547         8.28271
 12            9            1.80016         8.27434
 13            9            1.79056         8.28699
 14            9            1.78648         8.29067
 15            8            1.79567         7.41192
 16            8            1.80159         7.43702
 17            8            1.82159         7.46540
 18            8            1.79465         7.47846

MEAN           8.83         1.79809         8.14933
```

```
TABLESIZE:                  4999
NUMBER OF KEYS INSERTED:    4899
PERCENT OF TABLE FILLED:      98
DEPTH OF RECURSION:           10
PENALTY FUNCTION USED:    LINEAR

TRIAL     LONGEST PROBE     MEAN PROBES     MEAN REJECTION
-----     -------------     -----------     --------------
  1             7             1.75199          6.62482
  2             7             1.76546          6.58685
  3             8             1.76484          7.40579
  4             7             1.75688          6.56195
  5             7             1.77709          6.59951
  6             7             1.74321          6.59420
  7             7             1.76811          6.60318
  8             7             1.76219          6.59154
  9             7             1.77342          6.56031
 10             7             1.76260          6.58562
 11             7             1.75750          6.62318
 12             7             1.74239          6.60073
 13             7             1.76423          6.58215
 14             7             1.76832          6.58603
 15             7             1.77995          6.62237
 16             7             1.75423          6.59889
 17             8             1.75729          7.48867
 18             7             1.76382          6.58154

MEAN           7.11           1.76186          6.68874
```

The table was filled to 98% loading. All the keys in the table were retrieved to calculate retrieval performances. An equal number of keys not in the table were rejected to calculate rejection performances. 49% of the keys were deleted. Retrieval and rejection statistics were again calculated. The table was filled back to a 98% loading. Retrieval and rejection statistics were again calculated. These are the mean results of 18 trials:

TABLESIZE:	4999
DEPTH OF RECURSION:	4
PENALTY FUNCTION USED:	LINEAR

ACTION	# OF KEYS IN TABLE	% TABLE LOADED	LONGEST PROBE	MEAN PROBES TO ACCEPT/REJECT
INSERT 4900	4900	97.9	9.06	
LOOKUP 4900				1.80268 AC
LOOKUP-4900				8.35276 RJ
DELETE 2450	2450	48.9	8.61	
LOOKUP 2450				1.78899 AC
LOOKUP-2450				7.98171 RJ
INSERT 2450	4900	97.9	9.50	
LOOKUP 2450				1.86280 AC
LOOKUP-2450				9.43040 RJ

AVERAGES OF EIGHTEEN TRIALS

NBS-114A (REV. 7-73)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET	1. PUBLICATION OR REPORT NO. NBSIR 78-1551	2. Gov't Accession No.	3. Recipient's Accession No.

4. TITLE AND SUBTITLE ACCESS FUNCTIONS FOR PACKED SCATTER TABLES	5. Publication Date August 1978
	6. Performing Organization Code
7. AUTHOR(S) Bruce E. Martin	8. Performing Organ. Report No.
9. PERFORMING ORGANIZATION NAME AND ADDRESS NATIONAL BUREAU OF STANDARDS DEPARTMENT OF COMMERCE WASHINGTON, D.C. 20234	10. Project/Task/Work Unit No. 6401129
	11. Contract/Grant No.
12. Sponsoring Organization Name and Complete Address *(Street, City, State, ZIP)* see 9 above	13. Type of Report & Period Covered
	14. Sponsoring Agency Code

15. SUPPLEMENTARY NOTES

16. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here.)*

Three PASCAL access routines are given for packed scatter tables. INSERT packs tables of integer keys; FIND retrieves the keys; DELETE deletes the keys. While the routines currently access integer keys (for use in performance testing with pseudo-random integers), they can easily be converted to access other data types -- character strings in a symbol table, for example. To enhance portability, the code is straightforward PASCAL without any input/output capabilities. Appendices contain specific comments on the routines, listings, and sample results.

17. KEY WORDS *(six to twelve entries; alphabetical order; capitalize only the first letter of the first key word unless a proper name; separated by semicolons)*

Access functions; fast retrievals; hashing; PASCAL; scatter storage

18. AVAILABILITY [✓ X] Unlimited [✓ XX] For Official Distribution. Do Not Release to NTIS [ ] Order From Sup. of Doc., U.S. Government Printing Office Washington, D.C. 20402, SD Cat. No. C13 [ ] Order From National Technical Information Service (NTIS) Springfield, Virginia 22151	19. SECURITY CLASS (THIS REPORT) UNCLASSIFIED	21. NO. OF PAGES 29
	20. SECURITY CLASS (THIS PAGE) UNCLASSIFIED	22. Price